California State University, Northridge

Network Bias Indication Trainer

A thesis submitted in partial fulfillment of the requirements for the degree of

Masters of Science in Computer Science

by

Sina Hesari

May 2017

The thesis of Sina Hesari is approved by:

_____	_____
Prof. Steven Fitzgerald, D.Sc.	Date
_____	_____
Dr. Fares Hedayati, PhD.	Date
_____	_____
Prof. Vahab Pournaghshband, PhD., Chair	Date

California State University, Northridge

Acknowledgments

First and foremost, I would like to acknowledge Professor Vahab Pournaghshband for his amazing support over the past academic year. I am very proud to have worked with such an amazing and passionate researcher. I also would like to acknowledge my parents for their unparalleled support over the past few years. Their love haskept me going through many sleepless nights.

I would also like to thank Mr. Paul Kirth, Mr. Richard Dojillo and Mr. Aren Boghozian for working with me on this project. Lastly, a huge acknowledgement to the Medtronic family, especially Mr. Narek Isaghulyan who allowed me to work on a flexible schedule to finish this project.

Dedications

I would like to dedicate this work to my lovely parents. Their amazing love and support has made my life so much simpler. I also would like to acknowledge my late grandmother who left her country of birth to make sure I would get to the States safely. She passed as I was about to obtain my bachelor's degree. I am sure she would have wanted me to continue my education.

Table of Contents

Abstract

Network Bias Indication Trainer

By

Sina Hesari

Master of Science in Computer Science

Network Neutrality has been a hot topic since the proliferation of the internet. Indeed, there have been numerous efforts by the research community to expose the Quality of Service (QoS) policies that could lead to violations of net neutrality. This paper is building upon their success and is intended to employ new methods that can assist in detecting such violations.

There are many methods that an Internet Service Provider can implement to violate Net Neutrality. For the most part, we will focus on Strict Priority Queueing (SPQ). SPQ leaves a unique and interesting pattern in our detection packet trains. Our goal is to train a Machine Learning classifier that can identify whether a packet train has gone through a network that violates Net Neutrality using Strict Priority Queueing.

In this paper, we will employ statistical models and Machine Learning techniques to identify the areas where ISPs are violating Network Neutrality. Our goal is to show that with Machine Learning the detection of network neutrality will require smaller and

less detectable packets. Our hope is that researchers will employ more Machine Learning related techniques to identify Network Neutrality violations.

One of the main classifiers in Machine Learning is Support Vector Machines. We have decided to implement this thesis using an SVM identifier. SVMs are great at identifying division lines in binary data sets. Therefore, an SVM classifier can detect whether a certain condition exists or not. This is useful for our cause because our goal is to identify whether a packet train has gone through a network that discriminates using SPQ or not. Additionally, we will use Random Forest to train a second set of classifiers and compare their results.

My goal of writing this paper and doing a research in this field was not to make a political stand but rather to create transparency and provide more information to internet users.

Chapter 1: Introduction

Net Neutrality is one of the main topics of our generation. Net Neutrality represents an ideal world where the Internet Service Providers will treat all internet packets that are being transmitted as equals. When perfect Net Neutrality is being practiced, the popular contents will be transmitted at a faster rate compared to less popular, less in demand contents [1]. In this world, the users will dictate which services are supposed to get higher bandwidth [2]. The Internet Service Providers will only act as conduits and will not enact any policy to benefit their own bottom lines by treating packets differently.

Some ISPs can adopt policies that allocate more bandwidth to internet data that provides them with higher profits. For example, if an ISP owns a content provider, this ISP could give extra priority to the packets that are sent from their subsidiary. At the same time, if any other content provider wants to enjoy this level of priority, they would need to pay extra [3].

Indeed, actions like these can exacerbate the user experience. For example, if a user enjoys a TV show but the streaming service providing this TV show receives a lower priority from the ISPs, the user will not have a quality experience while watching that show [4]. Similarly, the buffering time will be more than services with higher priorities. This inconvenience can cause users to abandon a streaming service. Therefore, content providers with more capital will be able to afford paying off the extra fees for higher priority. The users and smaller content providers will be at a huge disadvantage in this

system. I would like to mention at this point that this paper is not a political paper. The goal of this research is not to decrease Net Neutrality, but rather to provide transparency on whether it exists [5].

Chapter 2: Discrimination Techniques

In this chapter, I will go over one of the policies in which Internet Service providers can violate Network Neutrality. These policies are also known as Quality of Service (QoS) policies. I will enumerate this method and its effects on internet packets. In this paper, I will describe the Strict Priority Queueing (SPQ) policy.

SPQ is a discrimination method where an ISP can assign two classes of priority to the packets. This means that certain data packets will have higher priority in transmission compared to the rest [6]. The ISP creates two storage queues in one of the network routers. One queue is dedicated to packets with high priority data and the other one is for low priority data. When packets enter this node, they will be assigned to their respective queue. To exit the router, the data from the high priority queue will be transmitted first. Therefore, the high priority queue must be empty before any of the packets from the lower priority queue can exit the router [3].

From a first glance, this might not seem intrusive. It may seem that the low priority packets will wait for a short time while higher priority data are being transmitted. However, in practice, this method can create substantial delay and loss [6]. To further argue this point, we define the idea of bottleneck [7]. Bottlenecks occur when the volume of data entering a node is larger than the data exiting this node. For example, if there is 10 Mb of data entering a node per second while only 2 Mb of data are being transmitted from it, the transfer of data through this node will slow down. One of the ways that bottleneck can occur is when there is network congestion during prime-time hours.

Let's look at a node in the network that implements the Strict Priority Queueing discrimination policy. During prime time or when there are bottlenecks in the network, we will have a backlog of data. If a router is implementing SPQ, then only the high priority packets entering this node can exit [8]. This means that the low priority data will be stored in the low priority queue and must wait until the high priority queue is completely empty. Meanwhile, after the low priority packet has been saturated, any additional low priority packets entering the node will be dropped. Consider the following example.

ISP XYZ has a contract with Streaming Service A to give its packets a higher priority compared to the packets form Streaming Service B. Two internet users are watching TV at home simultaneously during the rush hours of internet traffic. One user is watching Streaming Service A while the other one is watching Streaming Service B. Therefore, the Streaming Service B and Streaming Service A packets enter the discriminating node at the same time. When the Streaming Service A packets enter, they get stored in the high priority queue while the Streaming Service B packets get stored in the low priority queue. The Streaming Service A packets exit the node first. Because the users are watching their shows during prime time, more data packets are entering the node than exiting. Hence, while this first user is watching Streaming Service A, the second user must wait until all the Streaming Service A and other high priority packets have been cleared from the node. Only then Streaming Service B packets can be transmitted. And while the second user is waiting, any of the additional Streaming Service B packets entering that node will be dropped because the low priority packet is

full. This results into even more delays because the second user has to request the dropped Streaming Service B packets again.

The reason bottleneck is necessary for SPQ to take effect is that when data is being sent and received at the same speed, there will not be any backlog [7]. Therefore, the queues get filled and emptied in real time without causing any imposed delay or loss. The problem starts when network congestion develops and the lower priority packets have to wait until the high priority queue has been emptied out. After reading related research papers regarding Net Neutrality, I learnt that in order to detect violations, we often need to create congestion and backlog in networks [8] [7]. While creating congestion, we need to make sure we are not creating a negative effect for other users.

One of the challenges with creating bottlenecks is that the testbeds causing these congestions needs to send a large set of data packets [7]. The test bed needs to also know exactly how much data it should send. Therefore, we can potentially run an experiment and not be able to create the needed congestion. When the experiments used in this paper was run, a switch was used to ensure a bottleneck existed on path.

One of the downsides of this approach is that other people who want to run our experiment might not be able to create congestion on their own. The good news is that one of the upsides of using Machine learning is that we can detect Net Neutrality with smaller packet trains and shorter congestion periods. At the same, the users do not need to train their own classifiers. This idea and the potential benefits of ML algorithms will be discussed in later chapters.

In this paper, to train the Machine Learning classifiers, we focused on the SPQ packets. We chose one discrimination method because we wanted to first see how the Machine Learning classifiers can assist us in detecting Net Neutrality more effectively. By working on one method, we can pave the way to train classifiers for other QoS techniques. In addition, we chose SPQ because the detection packet trains create a detectable and unique pattern after going through a discriminating node. In the following chapters I will discuss how the data for detecting Strict Priority Queueing was designed and collected.

Chapter 3: Packet Structures

In this chapter I will go over the detection method used for Strict Priority Queueing. In this paper, we are using data collected by a testbed that was designed to detect SPQ by Pofessor Pournaghsband [9] [10]. An early version of the algorithm on how to detect SPQ was implemented by Pournaghsband [9]. In addition, an elaborate implementation of this testbed and SPQ detection was setup by Mr. Paul Kirth [7]. Their work [9] [7] provides great insight into discrimination methods and the different mechanism that can detect them. Since the focus of my paper is on Strict Priority Queueing, I will describe the SPQ detection method that was designed and developed by Pournaghsband [9] and further implemented by Kirth [7].

In this test bed, the data is being transmitted from a server source and will be received by a client. The testbed frame work is designed in C++ [7]. To achieve this goal, a set of virtual servers were used in Planet Lab. Planet Lab is a set of servers set aside by the academic and private sector communities [10]. Researchers can request to have access to these servers in order to run their experiments from multiple sources. Therefore, to run the experiment, the operator of testbed needs to choose certain Planet Lab nodes. The operator can then remotely log in to the selected Planet Lab nodes and install the proper files and software. Afterwards, the user can schedule certain experiments to be initiated from these nodes. A receiver client machine was set up at CSUN to receive and store this data [7].

To ensure the experiments were run properly, a hardware switch was setup between the sender servers and the receiver client machine in CSUN. This switch is a

Cisco Catalyst 3750 and has been configured with several discrimination policies including the Strict Priority Queueing [7]. This switch also created the much-needed bottleneck. The data from the Planet Lab nodes were sent with the speed of 10 Mbps. This switch allowed us to create a bottleneck of 2 Mbps. This implementation creates congestion. This bottleneck can also be used with other discrimination policies [7].

To detect Strict Priority Queuing, two different sets of packet trains are generated and sent from the Planet Lab nodes. The first one is a high priority packet train and the second a low priority one. In both packet trains, the testbed sends a predetermined number of high priority packets to saturate the high priority queue [8]. We will call these packets the saturation phase or saturation train. This way when the rest of the packets are sent, the high priority queue is full. In addition, the existence of a backlog will result to some loss [11]. Therefore, the mere existence of loss will not indicate the existence of Net Neutrality violations. What we look for here is the difference in patterns.

Let's look at the Low Priority Packet train first. After the saturation phase is over, the packet generator starts to generate the low priority phase. In the low priority phase, the server starts sending one Low priority packet in between a certain number of High priority packets [8]. The packets in between 2 low priority packets are called the separation packet train. This way we ensure that the high priority queue keeps getting filled up. Therefore, low priority packets have a lower chance of getting through the switch. This means that in theory, the loss percentage of Low packets will be much higher than High packets.

Next, we will discuss the high priority phase. In high priority phase, the sender server generates a saturation packet train similar to the low priority phase. Afterwards the

high priority phase gets generated by sending a high priority packet and then inserting a specific number of low priority packets in between them [8]. This is the exact opposite of what low priority phase did. In high priority phase, the high priority packets are separated by a certain number of low priority packets. Therefore, the total number of packets are the same in both high and low priority packet trains. Both saturation phases have the same number of packets [8]. In addition, the number of high priority packets in the high phase and the number of low priority packets in the low phase are also equal. Finally, the length of each separation packet train is equal as well.

As part of my project, I needed to understand the details of detection algorithm and its existing implementation. I found out that the saturation phases consisted of 1,000 high priority packets. In the low priority phase, a total 5,000 packets would be transmitted. We would have 1,667 low priority packets and 3,333 high priority packets would be sent in between them [7]. I have drawn Figure1 and Figure 2 to further illustrate this point. In Figure 1 and Figure 2, S represent a packet in saturation phase. H represents a High packet and L represents a Low packet.

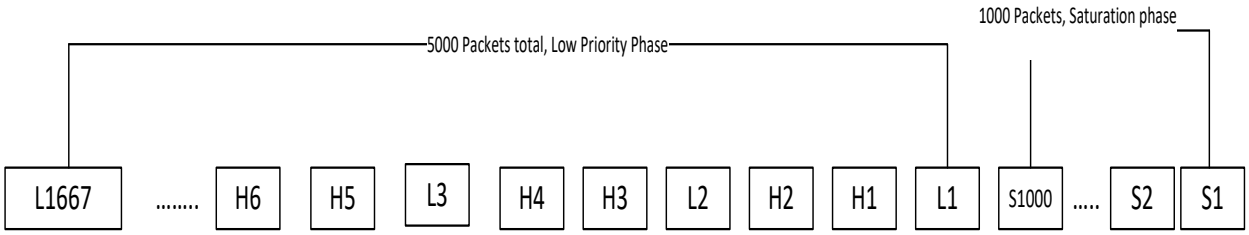

Figure 1: Low Priority Packet Train

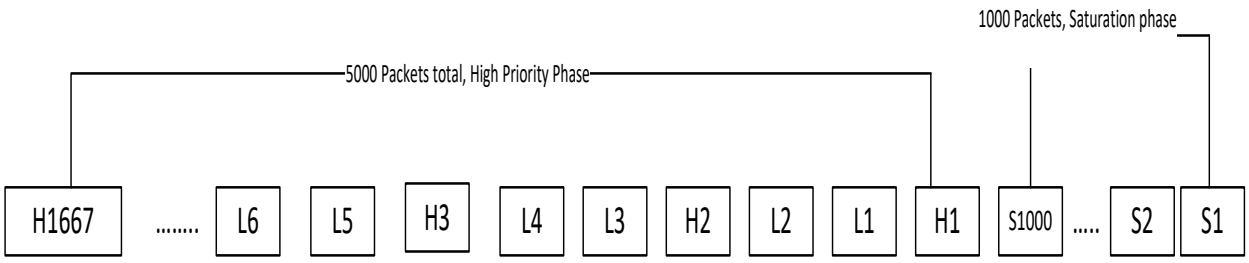

Figure 2: High Priority Packet Train

Let's look at the Streaming Service B vs. Streaming Service A example. We are going to create our packet trains using Streaming Service A and Streaming Service B packets. The ISP XYZ is giving a higher priority to Streaming Service A packets. A low priority packet train would start with 1,000 Streaming Service A packets. Next, the packet generator node would send the first Streaming Service B packet. This packet would be packet number 1. For each Streaming Service B packet sent, the test server would generate two Streaming Service A packets. Thus, the first two Streaming Service A packets are packet number 2 and packet number 3. This process would repeat until packet number 5,000 was sent. A similar situation would repeat for the High priority packet train.

The rationale behind creating these two packet trains is to compare their loss percentage. In Pournaghsband's algorithm [9], the received packet trains would be cleaned up. The saturation phase would be scrapped and all the low priority packets in the high priority phase would be filtered out. The tester would only need to look at the high priority packets in the high priority phase. Similarly, the saturation phase and the high priority packets would be omitted in the low priority phase.

At the same time, both packet trains are sent from the same source and right after one another [7]. This means that both packets took the same path to our receiver machine. Hence, we can compare the loss of low priority packets in the low phase with the loss of high priority packets in the high phase. The difference between their loss rate can determine whether SPQ exists on the path. This calculation does not make any decision based on the existence of loss but rather focuses on the loss difference between the two priority levels. In theory, we expect that the high priority packets would have a lower loss percentage compared to low priority packets.

The reason I want to work on this area is to develop a classifier that detects SPQ policies without looking at the aggregate values of the entire packet trains. One of the downsides of the previous method is that two large packet trains are required to be sent. Each packet train is about 6 Megabytes. Sending two packets would use 12 Megabytes of data [7]. This experiment needs to be run multiple times to confirm the results. Our goal is to decrease this size and make the testbed less intrusive on the network.

Indeed, sending large packets of data can increase suspicions for the ISP. If the ISPs become aware of the types of data we are sending to test their neutrality, they can meddle with our findings [5]. Therefore, we have additional incentive to make our packets as small as possible. I believe the feature selection of Machine Learning can help us with creating smaller packets. In fact, we might even be able to send one packet train instead of two. This improvement will help the testbed run faster experiments while imposing less intrusion on the network. We will discuss how to improve our test bed in future chapters. I will enumerate my findings and suggestions regarding the testbed.

Chapter 4: Working with Data

In this chapter, I will go over the data that I used for this experiment and how we prepared our data to be trained by our classifiers. One possible option was to run the experiment from scratch. A few items need to be setup before running the testbed.

First, a slice of Planet Lab nodes need to be assigned to the tester. The tester can then choose specific servers in this slice to run the experiments from them [7]. Next, the tester needs to parallel SSH into these server nodes and install the applications and libraries required to run the testbed. Additionally, the tester needs to install the testbed code in Planet Lab nodes. The tester should also schedule the experiments to be run and sync the server nodes with the receiver machine. Finally, the tester needs to connect the servers to a MySQL database in order to collect data [7].

I was able to run most of these steps; however, the quality of Planet Lab nodes have decreased recently. This loss of quality prevented me from running the experiment. In addition, I could not connect the servers to the MySQL database and start collecting data. I will go over my findings about the testbed, and how we will improve it in future chapters.

Thus, I decided to use existing data from previous experiments. I used the data Mr. Kirth had collected using the testbed [7]. This data is still new and relevant to our work. The tests were run in April 2016 and the Strict Priority Queuing was implemented as described in chapters before. Pournaghsband [9] looked at aggregate loss and aggregate delay and used these data to infer if there is any discrimination. My goal was to

understand this data and find all patterns that could be used by Machine Learning algorithms.

The existing data was in the PCAP format. Each PCAP file contains all packets that had been received by the receiver machine. For example, if a packet train with 10,000 packets was sent and 4,000 packets were dropped, the PCAP file would show the remaining 6,000 packets that were received. Each PCAP file had the sender and the receiver information. This is very useful because the packet trains were sent from many different Planet Lab nodes.

There has been other research done regarding Net Neutrality around the world. Their goal is to indicate which regions of the world violate neutrality the most [10] [11]. By using Planet Lab nodes from around the globe, we accumulate diverse and universal data about loss and delay. My first goal was to ascertain which parts of data belonged to the saturation phase, high priority phase, and low priority phase.

I went over the C++ files implemented by Kirth [7] as well as a few other files on the testbed server to find this information. During my initial analysis, I realized that the packet trains would start with a random source port during the saturation phase. They would switch to source port 20000 when the priority phrase started. The packets that were sent to destination port 22222 were high priority packets. Originally, destination port 22223 was set for the low priority packets. However, while running the testbed this port was changed to 44444. The reason behind this change is that Planet Lab nodes had some issues generating the low priority packets for port 22223 [7]. Therefore, I used only the packet trains that were sent to destination port 44444. I will discuss the limitations of Planet Lab nodes in my findings. I will also indicate what future users of this testbed can

do to work around these limitations. For the SPQ files I used files 15109.pcap through 15286.pcap. The SPQ packets had a separation packet of size 2.

Each data packet has a packet number and a frame number. The packet number indicates the order in which this packet was generated and sent from the Planet Lab node servers. The frame number relates to the order in which the packets were received. For example, if a packet was sent from a Planet Lab node as packet number 1, but it was received by our receiver server as frame number 200, we can conclude that there was a delay during transmission of this packet. Therefore, 199 other packets were received before packet number one had arrived. Similarly, if a packet number is not in our PCAP file, we know that this specific packet was lost during transmission. This numbering can give us a big advantage in finding patterns for our machine learning algorithms. For more information refer to Figure 3.

2025	PType	PID	Timestam	FrameNumb
2026	L	1025	31827.01	2025
2027	L	1026	31831.12	2026
2028	H	1027	31840.71	2027
2029	L	1028	31871.14	2028
2030	L	1029	31877.23	2029
2031	H	1030	31899.55	2030
2032	L	1031	31915.3	2031
2033	L	1032	31919.4	2032
2034	H	1033	31944.7	2033
2035	L	1034	31973.6	2034
2036	L	1035	31977.7	2035
2037	H	1036	32003.52	2036
2038	L	1037	32045.17	2037
2039	L	1038	32049.69	2038
2040	H	1039	32074.54	2039
2041	L	1040	32089.77	2040
2042	L	1041	32093.87	2041
2043	H	1042	32107.5	2042
2044	L	1043	32134.38	2043
2045	L	1044	32138.49	2044

	PType	PID	Timestam	FrameNumber
20	H	2582	12803.15	19
21	H	2996	12807.25	20
22	H	3348	12811.77	21
23	H	3749	12815.87	22
24	H	4188	12820.38	23
25	H	4650	12824.49	24
26	H	5060	12828.59	25
27	H	5570	12833.1	26
28	H	5966	12837.21	27
29	H	6473	12841.72	28
30	H	6902	12845.82	29
31	H	7349	12850.34	30
32	H	7821	12854.44	31
33	H	8237	12858.95	32
34	H	8693	12863.06	33
35	H	9149	12867.57	34
36	H	9656	12871.67	35
37	L	1	12876.18	36
38	L	10	12880.29	37
39	L	28	12884.8	38
40	L	49	12888.9	39

Figure 3: An example of two packet trains. The packet train to the left has low loss and the packet to the right has experienced high loss.

Next, I used a Python script to convert the PCAP files and extract the proper data from them. I collaborated with Mr. Richard Dojillo in developing this script. We incorporate the os and csv libraries of Python in our scripts. We start by converting each PCAP file into a TXT file. The text file has all the packets that have arrived with their respective data. Next we did some testing to find our desired parameters in each packet. We decided to keep the packet priority, sender packet ID, time of arrival and the receiver packet frame of each packet. We saved this information in a CSV file. Therefore, each PCAP file was transformed into a CSV file that made our desired data more accessible.

After consulting with Doctor Hedayati and Professor Pournaghshband, I decided to extract some features from these newly generated CSV files. While incorporating machine learning, it is imperative to have all the data points in one file. Since we have thousands of packets in each packet train, we decided to break them down to facilitate the feature selection. While analyzing the data, I realized that the loss and delay patterns of Strict Priority Queueing packets would fluctuate throughout the PCAP file. Therefore, we decided to break each of the packet trains into 50 sections.

In each section, we decided to look at certain characteristics. We calculated the percentage of high and low packets in each section. Next, we kept track of the delays of high priority and low priority packets. We calculated the average and variance of the delays for each section. In addition, we looked at different percentiles of delay in each section.

While analyzing data, I noticed that in the low priority phase there are usually many high priority packets in between two low priority packets. Thus, we calculated the number of high packets that were received consecutively. We recorded the consecutive

number of high packets in a list and calculated their average, variance, and percentiles. After all the calculations were done, we generated 2,100 features for each PCAP file. A classifier such as Support Vector Machine can be trained by these features to make a decent prediction about the existence of Strict Priority Queueing.

Table 1 represents the 42 features that were extracted in each of the 50 packet train sections. In Table 1, H represent High Priority and L repents Low Priority.

Feature Number	Feature	Feature Number	Feature
1	Percentage of H packets in the Packet Train Section	22	40th percentile of delay for L packets
2	Percentage of L Packets in the Packet Train Section	23	50th percentile of delays for L packets
3	Average delays for H packets	24	60th percentile of delay for L packets
4	Variance of delays for H Packets	25	70th percentile of delays for L packets
5	Minimum delays for H packets	26	80 percentile of delay for L packets
6	10th percentile of delays for H packets	27	90 percentile of delays for L packets
7	20th percentile of delay for H packets	28	Maximum delays for L packets
8	30th percentile of delays for H packets	29	The number of H packet trains in a row
9	40st percentile of delay for H packets	30	Average size of the H packet trains in a row
10	50th percentile of delays for H packets	31	Variance size of the H packet trains in a row
11	60th percentile of delay for H packets	32	Minimum size of the H packet trains in a row
12	70th percentile of delays for H packets	33	10th percentile size of H Packet trains in a row
13	80 percentile of delay for H packets	34	20th percentile size of H Packet trains in a row
14	90 percentile of delays for H packets	35	30th percentile size of H Packet trains in a row
15	Maximum delays for H packets	36	40th percentile size of H Packet trains in a row
16	Average delays for L packets	37	50th percentile size of H Packet trains in a row
17	Variance of delays for L Packets	38	60th percentile size of H Packet trains in a row
18	Minimum delays for L packets	39	70th percentile size of H Packet trains in a row
19	10th percentile of delays for L packets	40	80th percentile size of H Packet trains in a row
20	20th percentile of delay for L packets	41	90th percentile size of H Packet trains in a row
21	30th percentile of delays for L packets	42	Maximum size of H Packet Trains in a row

Table 1: List of features in each section

4.1 Feature 1: Percentage of High Priority Packets

In each packet train section, we received a number of high priority and low priority packets. Our goal was to look at the number of high priority packets and compare them to the total number of packets in that packet train section. This calculation gives us the percentage of the packets in this section that were high priority. The percentage of high priority packets can point to the loss characteristic of the packet train section. This way, we can detect whether this percentage fluctuates between data points and throughout the packet flow.

4.2 Feature 2: Percentage of Low Priority Packets

Similar to the first feature, we wanted to know what percentage of the packets were low priority. Therefore, we divided the number of low priority packets by the total number of packets in each section. This calculation can guide us in tracking loss throughout the packet train and in between data points.

4.3 Features 3&4: Average and Variance for High Priority Delays

In each section, we take a look at the delay of high priority packets. The delay values are stored in a numpy array. At the end of each packet train section, we calculate the average and variance of the delays for high packets. After the average and variance, we find the percentiles related to these values.

4.4 Definition of Percentiles

A percentile represents a point in a series where a certain percentage of the values in the series are less than this point. Therefore, the Nth percentile is the value where N% of the data points are behind it. For example, the 0^{th} percentile is the lowest value in a series. The 50^{th} percentile is the median value of a list. The 100^{th}

percentile is the highest value in a list. And 30^{th} percentile is the value where 30% of our data are less than this value. For example, if our list is the numbers 1 through 10, the 0^{th} percentile is 1, and the 100^{th} percentile is 10.

4.5 Features 5-15: The Percentile Values for Delays of High Packets

As mentioned earlier, the delays for High Priority Packets are stored in an array. We can then look at the values in this array and calculate what the values for certain percentiles are. We calculated 11 percentiles for these features. We started with the lowest value in the series or the 0^{th} percentile and calculated all percentiles that are a factor of 10. This way we would collect a representative distribution of the delays for high packets in each packet train section. Our goal was to find out whether the delay distribution would change between the baseline and SPQ data.

4.6 Features 16-28: The Low Priority Packet Delays

Next we put all the delays for the low priority packets in an array and calculated their average, variance, and percentiles. Our calculations were similar to the ones performed for the delays of high priority packets. Our goal was to find whether the averages and the distributions change throughout the packet train and between baseline and SPQ data.

4.7 The Number of Hight Priority Packets in between two Low Priority Packets

While looking at the data, I noticed that the SPQ packet trains might have a few high priority packets in a row. Since this is a big deviation from the original patterns, I decided to incorporate this discrepancy in our features. Therefore, in each section, we recorded how many high priority packets existed in between two low priority packets. I would set a counter for high priority packets and each time a low

priority packet appeared, I would record that number in a list and set the counter to zero. For example, if the pattern was “Low High High Low Low High Low”, I would insert 2,0, and1 into my list. My goal was to find an interesting pattern for SPQ detection.

4.8 Features 29-42: The Average, Variance and Percentiles of High packets in a row

After creating the list from section 4.7, we started to look at the features we could extract from it. We decided to extract the size of this list as one of our features. This feature can tell if the high priority packets were spread out or concentrated in one or two groupings. Next, we calculated the average, variance and the respective percentiles for this list. Our goal was to compare these values and give the learning algorithm an opportunity to identify the sections that had long consecutive high packets.

4.9 Baseline Data

So far we have cleaned up all the data where Strict Priority Queueing exists. These are our positive cases. The classifier needs to have some negative cases as well. Otherwise, it will not be able to make proper predictions. Henceforth, we refer to the negative cases as baseline. Since no specific baseline data was run with the SPQ packet trains, I used a similar set of data that experienced the same exact bottleneck value as SPQ data did. For the base line files, I used files 15752.pcap through 15951.pcap.

The data we used consisted of one set of packets and was not broken into low and high packets. This characteristic does not create a problem for our baseline data. The main value that needs to be the same between our SPQ and baseline data is the size of the bottleneck. In addition, the data we used was run through a node with only one queue.

This is the same characteristic needed for running the baseline packet trains. However, the baseline data needs to be cleaned up and transformed.

The approach we took was to use the packet ID numbers. We already knew which packet numbers would be high and which packets would be low in a SPQ packet train. Therefore, I developed a script that would transform the baseline PCAP files and assign high and low values to their individual packets. I created two similar scripts. One script generates high priority baseline packet trains while the other one generates low priority baseline packet trains. I made sure to create the high and low baselines from separate files. The baseline packet trains were converted into CSV files. We extracted the same 2,100 features from the baseline files as we did from the Strict Priority Queueing files. However, there is one difference between the baseline and SPQ features.

The features of each PCAP file are stored in one row in a new CSV file. This CSV file will be used to train our classifier. Each column represents one of the 2,100features. The classifier needs to be able to distinguish between the baseline and SPQ files. Therefore, each row gets one additional column. This new column will represent whether Net Neutrality violations were present or not. Therefore, for this feature, the SPQ files will get a value of 1 and the baseline files will get a value of 0. Henceforth, we will call this file the CSV Test Points File.

	A	B	C	D	E	F	G	H	⋯	CBU
1	1	0.977273	0	94.18821	2859.239	0	22.157	40.2106		157
2	1	0.978723	0	86.7155	2863.936	0	17.0871	34.167		157
3	1	0.980392	0	279.2639	47359.69	0	25.853	51.698		157
4	1	0.978261	0	89.70393	2803.289	0	18.0536	35.8618		157
5	1	0.995434	0	16.15571	1675.534	0	0.1757	0.3644		157
6	1	0.995434	0	0.954586	0.29662	0	0.2029	0.3926	⋯	157
7	1	0.977273	0	81.3266	3536.891	0	0.0098	16.408		157
8	1	0.977273	0	77.33823	2853.549	0	13.124	25.85		157
9	1	0.977778	0	80.47744	2692.505	0	16.4127	29.9534		157
10	1	0.977778	0	96.39973	2990.998	0	22.5719	41.0358		157
11	1	0.978723	0	100.6238	3270.787	0	23.381	42.663		9
12	1	0.980392	0	89.82734	3738.457	0	17.233	31.594		9
13	1	0.981132	0	93.88993	4370.553	0	16.7788	29.9614		12
14	1	0.978723	0	87.91735	3727.992	0	12.7161	23.7948		13
15	1	0.957447	0.021277	81.34009	2941.465	0	16.0011	30.1108		13
⋮			⋮			⋮		⋮		⋮
376	0	1	0	1.015467	0.481153	0	0.1654	0.3284		
377	0	1	0	24.82781	925.5832	0	0.3666	0.9972	⋯	33

Figure 4: The CSV Test Points File. Each row represents one PCAP file or one experiments. The first column represents the classification of data. SPQ data have a classif of 1 and BaseLine data have a classif of 0. The remaining columns represent the 2100 features

To test our classifier, we separated some data points from the training set. The classifier will be tested by the data points not in the training set. This approach will rid our classifier from potential biases of training and testing the classifier with the same data points. Python has a function that can provide this capability for us.

Chapter 5: Machine Learning Algorithms

One of the major parts of any Machine Learning project is the Feature Selection and Feature Engineering [12]. Therefore, the emphasis of this project is on gathering data and working to extract the proper features needed for our algorithms. Indeed, data scientists spend more time on engineering and optimizing the features rather than running their algorithms. This is because most Machine Learning algorithms have already been implemented and the fact that some level of human genius is needed in extracting the features from data [12].

The algorithms I choose must be great at classifying data. In addition, these algorithms should be able to work with our training sample size. The first algorithm we considered is Support Vector Machine (SVM). SVM is a supervised learning classifier and is designed to calculate a fine line between positive and negative data [13]. SVM was created by Mr. Vladimir Vapnik. SVM works in vector space. Each data point receives an X and a Y value. The X values represent the features of our data and the Y value represents the classification of this data point. In other words, Y represents if data belongs to the positive or negative group.

Therefore, a data point can be defined by multiple features and belong to one of two groups. The Support Vector Machine algorithm will consider all the feature values and draws the data points on a graph. At this point, the SVM will look for a distinction line between the two sets of data. This line divides our training sample into two separate sections of positive and negative data points.

The separation line must have certain characteristics. First, our separation line needs to divide our training sample into the best two sections possible. This means that the distinction groups need to be optimal. At the same time, there has a to a buffer from both sides of the line where the closest positive and negative values need to a have a minimal distance from the separation line [13]. This section is called the street and two line with the minimum distance from the middle of the street are called the gutters. The goal of SVM is to find the best distinction line where no data point is between the middle of the street and the two gutters. For simplicity, let's assume that the feature set in X can be represented on a two-dimensional plane. In addition, if the Y value is positive our data point will be black and if the Y value is negative the data point will be white. The result is a two-dimensional graph with all of data points plotted out. We can visually see where these points lie and what their respective Y values are. The following figure will give an example of this representation of data.

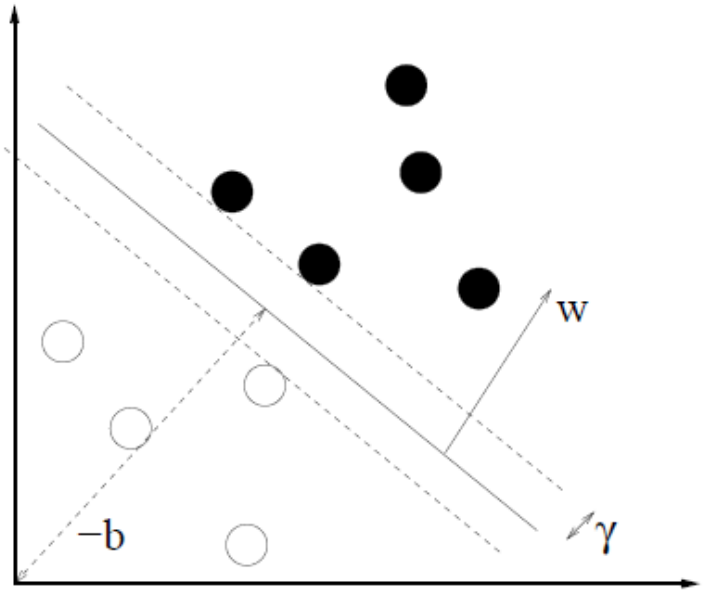

Figure 5: An example of an SVM Classifier *[14]*

In this figure, W and -b are the separating hyperplanes. Gamma is our margin. The solid line is the middle of our street and the two dotted lines are the gutters. This classification divides the graph in two different sections where there is a good margin between the closest data points and the middle classifier line on both sides. The goal of an SVM algorithm is to train the classifier in finding the most optimal line in the data set. Afterwards, the classifier can receive new data and determine where they would be positioned on this graph. This way the classifier can make an educated prediction as to whether this data belongs to the positive or negative group.

I did not implement the SVM on my own. The sklearn [15] and scipy [16] libraries of Python have already implemented the Machine Learning algorithms I need. While running my program, I realized that the sklearn [15] and scipy [16]libraries are not compatible with the Windows operating system. Thus, I switched to an Ubuntu machine. I will discuss the actual results I obtained in the next section.

After getting some promising results with Support Vector Machine, we decided to train a Random Forest Classifier with our data. Random Forest Classifier is an ensemble of decision trees that grow and split based on the features in our dataset [17]. The algorithm creates many subsets of the original data. The algorithm then creates decision trees with the subset of our data. The Decision Trees are similar to the tree data structures [18]. However, in decision tress, at each node, the splits are done based on the values of our features. For example, if one of our features is the percentage of High packets, the node's left child could represent the less than 50% and the right child could be more than 50%.

Next, we will make our decisions based on the accuracy of the generated trees. Because we are creating and ensemble of trees, each decision tree votes on the effectiveness of the features it used. The features with the highest value are most effective to our decision making. Therefore, the classifier will give more weight to those features. In the end, the most important features are used in our classifier. Random forests have low variance and do not overfit the data because the classification is done with breaking down the data into smaller samplers rather than looking at the entire training data. For this reason, Random Forests are widely used and are accurate [17].

In addition, the process of finding the more important features is useful because we can access the most important ones and ascertain more information about the data. For example, if all the important features are in the first 20% of our packet trains, we might be able to omit sending larger packets. Additionally, if the Random Forest classifier is not giving us the optimal solution, we can keep the most desirable data and continue with our feature engineering.

Chapter 6: Results

In this chapter, I will go over the results I found with my classifiers. My CSV Test Points File has 377 rows and 2101 columns. Each row represents data form one experiment or one PCAP file. Additionally, each column represents one feature. My classifier takes all the data and features in the data points file and reads them into a 2-dimensional Python numpy array [19]. Next, this numpy array is shuffled like a deck of cards to make sure the ordering of my original PCAP files do not affect the outcome. This numpy array then gets separated in two matrices. The first matrix represents the features. Therefore, it is a matrix with 377 rows and 2,100 columns. The second matrix represents the class sections which only contains values of either 1 or 0. Therefore, the class section is a matrix of 377 rows and 1 column. I will then divide my two matrices into a training set and a test set. This division is done randomly. Therefore, each time I run the experiment, I will get a new classifier. To implement this process, I used the train_test_split() [20] API call in Python. The train_test_split() takes the X and Y sets, and accepts a split percentage [20]. The split percentage decides the percentage of data that is dedicated to our test set. The remaining will be given to the training set. This API call will then return the training and test sets.

API Call	Library
train_test_split()	sklearn.model_selection
LinearSVC()	sklearn.svm
fit()	sklearn
predict()	sklearn
accuracy score()	sklearn.metrics
confusion matrix	sklearn.metrics
RandomForestClassifier()	sklearn.ensemble
feature_importance_	sklearn.ensemble

Table 2: List the API calls used for this experiment

For my tests, I dedicated two thirds of my data to train the classifier and one third of data to test the accuracy of this classifier. The reason I chose this division is based on some experiments I ran. My classifiers had the most consistency when the size of my test set was 33% of the original data set.

After obtaining the two data sets I used the svm.LinearSVC() [21] and fit() [21] API calls to train my classifier. The svm.LinearSVC() function creates a classifier and the fit() function trains the classifier using the training set. The classifier was then tested with the predict() [21] API against the test set. The predict() function takes the test set and runs it against the classifier. I compared the results given to us by the predict function with the actual Y set of the test set. To verify my results, I used accuracy_score() [22] and confusion_matrix() [23] API calls. The accuracy_score() function will return how often the classifier was right in its predictions and the confusion_matrix() fucntion will provide us with the false negatives and false positive values. The results of both accuracy_score() and confusion_matix() were consistent with each other.

At this point, I decided to run the experiment for 100 times and get the average accuracy. I am happy to announce that my classifier had an average accuracy of 92% and a median accuracy of 94.4%. In addition, the accuracy variance is only 1%. This result confirms our original idea that dividing each packet train into multiple sections will result in an accurate classifier. I chose to run my experiment for 100 times to have a basis for my average accuracy. Moreover, I chose to display both average and median accuracy to demonstrate that there was not a large gap in between them and the average and median accuracies were close to each other.

Next, I ran this experiment for 1,000 times and the results stayed relatively unchanged. The average accuracy went down by 0.3% to 91.7%. Median accuracy fell to 93.6%. However, the variance also went down to 0.7%. Finally, I ran this experiment 1,500 times and the values did not change. I believe this consistency in high accuracy is an evidence that my model is solid for detecting network neutrality. I ran the experiment for 1,000 and 1,500 times to demonstrate that the average accuracy did not plummet and stayed the same. We can conclude from this experiment that the average accuracy of 91.7% is representative of our SVM classifier.

After getting some promising results with SVM, I decided to run my experiment with Random Forests. I imported the RandomForestClassifier [24] from the sklearn.ensemble [25] library package in order to run my experiment with random forests. Afterwards I used the RandomForestClassifier() [24] API call to train my new classifier. With random forests, I needed to figure out how many estimator sets the algorithm needs to create. I started with 1,000 estimators and the algorithm started returning an average accuracy of 96%. Then, I increased the number of estimators to

10,000. I immediately noticed that the training time for my classifier increased. At the same time, my accuracy percentage increased as well. I ran the experiment 50 times to get the new average accuracy.

This experiment lasted about 60 minutes but we got some great results. The average accuracy climbed to 99% and the mean accuracy soared to 100%. After obtaining such promising results, I decided to run another test to find out which features are most important to my classifier. The results of this experiment can demonstrate whether we can reduce the packet train sizes. To get the top features, I employed the feature_importance_ [26] API call. feature_importance_ gives us the importance of each features used in our experiment. In addition, to rank the features based on their importance, I used the argsort function of numpy library. Table 2 lists the top 30 most important features that my random forest classifier employs. In Table 2, H represents High priority and Low represents Low priority.

Feature	Importance
1. Packet Train Section 1: Percentage of H Packets	1.79%
2. Packet Train Section 1: 20th Percentile for H Packet Delays	0.97%
3. Packet Train Section 43: 80th Percentile for H Packet Delays	0.94%
4. Packet Train Section 6: 90th Percentile for H Packet Delays	0.83%
5. Packet Train Section 49: Max Number of H Packets in between L Packets	0.79%
6. Packet Train Section 1: 10th Percentile of H Packet Delays	0.79%
7. Packet Train Section 24: Max H Packet Delays	0.74%
8. Packet Train Section 38: Maximum Number of H Packets in between L Packets	0.73%
9. Packet Train Section 20: Maximum Number of H Packets in between L Packets	0.71%
10. Packet Train Section 9: Variance of H Packets in between L Packets	0.70%
11. Packet Train Section 23: Variance of H Packets in between L Packets	0.68%
12. Packet Train Section 44: Maximum Number of H Packets in between L Packets	0.68%
13. Packet Train Section 39: Average of H Packets in between L Packets	0.68%
14. Packet Train Section 12: Variance of H Packets in between L Packets	0.68%
15. Packet Train Section 22: Variance of H Packets in between L Packets	0.68%
16. Packet Train Section 34: Maximum Number of H Packets in between L Packets	0.67%
17. Packet Train Section 35: Variance of H Packets in between L Packets	0.67%
18. Packet Train Section 15: Maximum Number of H Packets in between L Packets	0.67%
19. Packet Train Section 11: Varinace of H Packets in between L Packests	0.67%
20. Packet Train Section 25: Varinace of H Packets in between L Packests	0.66%
21. Packet Train Section 27: Maximum Number of H Packets in between L Packets	0.66%
22. Packet Train Section 37: Variance of H Packets in between L Packets	0.65%
23. Packet Train Section 7: Maximum Number of H Packets in between L Packets	0.65%
24. Packet Train Section 24: Variance of of H Packet Delays	0.65%
25. Packet Train Section 5: Variance of H Packets in between L Packets	0.64%
26. Packet Train Section 6: 40th Percentile of H Packet Delays	0.64%
27. Packet Train Section 44: Variance of H Packets in between L Packets	0.64%
28. Packet Train Section 44: Maximum Number of H Packets in between L Packets	0.63%
29. Packet Train Section 43: Maximum Number of H Packets in between L Packets	0.63%
30. Packet Train Section 26: Variance of H Packets in between L Packets	0.63%

Table 3: The 30 Most Important Features for Strict Priority Queueing

In Table 2, the first column is the packet train section number and its sub-feature that Random Forest deemed important. The second column is the percentage of time that this feature was used for decision making in the Random Forest classifier.

As we can see, most of the top 30 features belong to the Maximum and Variance of H packets in between L packets. Let's reiterate what these numbers represent. When extracting our features, we decided to look at the number of H packets that are received in between two L packets. For example, if there were 100 packets in a packet train section and the first 3 packets were H, we would insert 3 in a list. Let's assume the next 10 packets were L followed by 5 H packets. In this case, we would insert ten 0's and one 5 in the list.

I only displayed the top 30 features because at this point the aggregate importance is almost at 25%. Also, The importance in the remaining features after this point declines. In addition, my goal was to indicate that we might be able to cut the packet sizes in half because most of the important features are in sections 1 through 25. In future projects, we can ascertain whether this thinking is correct or not.

I have summarized our findings in Table 3.

The Learning Algorithm	Description	Number of experiments	Average Accuracy	Median Accuracy	Training Time
Support Vector Machine	Linear	100	92%	94%	5 Seconds
Support Vector Machine	Linear	1000	91.70%	93.60%	5 Seconds
Support Vector Machine	Linear	1500	91.7	93.6	5 Seconds
Random Forest	1,000 Trees	50	96%	98%	10 Seconds
Random Forest	10,000 Trees	50	99%	100%	70 Seconds

Table 4: The Machine Learning experiments and their aggerate results

From Table 3, we can conclude that the Random Forest classifier with 10,000 trees has given us the most accurate result. I believe this will be the algorithm to use for a commercial tool that can detect Net Neutrality in real time. Since this classifier has a training time of 70 seconds it can be trained offline and then used many times

by users. Therefore, we can either train the classifier on our own server and deploy it to our clients, or have them run it for the first time and use it every other time they want to test their network. To compare our method with DiffProbe, the highest detection accuracy they obtained was 98.5% [6]. Our conclusion is that our detection is better not only because it is marginally higher but because our feature selection is superior. To detect SPQ, DiffProbe only considers the aggregate delay [6].

Another study that used real internet experiment data, ShaperProbe [27], was able to detect the Shaping policy with up to 95% accuracy. Even though, we don't detect for Shaping, our result could point to the fact that ShaperProbe could benefit from incorporating Machine Learning in their detection and analysis.

Chapter 7: Related Work

In this chapter I will go over the related work that helped guide my research. Mr. Paul Kirth's thesis gives an in-depth overview of the testbed and its implementation [7]. In that paper, Mr. Kirth covers the technical characteristics of the testbed and how the data packets were setup to detect different discrimination policies. In my thesis, I have presented an in-depth analysis of Mr. Kirth's detection packet train setup for SPQ. My work provides more documentation on how this process was setup. Mr. Kirth, performed some analysis on the collected data to discover which time periods during the day cause more loss. My analysis differs from Mr. Kirth's because I looked at each actual data point and extracted the features that can be used for Machine Learning analysis. In addition, I have considered finding different methods of reducing the packet trains that were generated by Mr. Kirth's testbed. My method has a shown a 99% detection accuracy on data collected from the testbed which was implemented by Kirth [7].

Detecting General Network Neutrality Violations with Causal Inference [1] describes the implantation of Network Access Neutrality Observer (NANO) system [5]. The Detecting General Network Neutrality paper distinguishes the difference between active and passive probing [1]. One of the main features of this paper is that it does not focus on one specific detection method. Neither does it focus on discrimination against a specific application. Instead, this paper looks at groupings of data and attempts to analyze the performance of each ISP compared to the average performance of the rest of the ISPs. They used a black-box statistical approach called Causal Inference. The causal inference setup has some similarities to the feature selection used in my paper.

In both approaches, the data is being broken into separate groups. This is done to improve the detection accuracy [1]. However, one disadvantage that Causal Inference has compared to Machine Learning is the need to enumerate all confounding variables [1]. If some of the variables that impact the outcome are missing from the analysis, Causal Inference will begin to perform poorly. In addition, finding all confounding variables can be very time consuming. On the contrary, Machine Learning classifiers can look at the given features and even identify the more relevant ones. If some features have not been identified, the classifier will still be able to make proper predictions.

Diffprobe is another paper that considers the detection of Net Neutrality discriminations policies. One of the methods studied here is Strict Priority Queueing. To detect SPQ, Diffprobe sends two different data flows [6]. The first flow is from a specific application that the ISPs discriminate against. The second flow is the probing flow and is similar to the application flow in order to receive a homogenous treatment. To detect SPQ, the delay values of application and probing flows are compared.

Diffprobe also uses the Kullback-Leibler divergence test [6]. In the KL test, the probability mass functions of both flows are estimated. The KL test then calculates the divergence of these two function and finds a null hypothesis for the detection of SPQ. KL test uses the logarithmic difference of the two functions to calculate its measurement. KL test is a great statistical tool; however, this paper only looks at the delay values of data packets. My calculations are different than this approach because they take more variables into account and look at different sections of each data.

Network Neutrality Inference [4] is a new approach for detecting Net Neutrality. Network Neutrality Inference looks at different paths in one network and attempts to find

the links that impact performance. This technique is similar to NANO where the performance of one ISP was compared to other ISPs [5]. In comparison to NANO, Network Neutrality Inference looks at the network topology and attempts to find the specific path or paths that violate neutrality.

This approach uses a classifier. A series of solvable equations are created for the network and their algorithm calculates the insolvability of each path. If the path is highly unsolvable, then this path is violating net neutrality. I believe our SVM classifier is a better indicator compared to their system. In addition, all their experiments were run through a network simulator and they did not test their theory on real internet experiment data. Our calculations are more reliable and only use real internet data.

Chapter 8: Future Work

I believe the first step to build on my work is to detect Policing and Shaping policies with Machine Learning algorithms. We already have enough data on both policies and the only reason I did not train a classifier for these policies was lack of time. A new researcher needs to study these policies and the detection packet trains our test bed employed. Next, the PCAP packet trains need to be converted into CSV and the features would be extracted from them. I believe that this project can be done in 5 months.

During the past academic year, I worked with a testbed that was already in place. While working on the testbed I noticed a few areas where we could see improvement. First, there are no clear documentation on the C++ code that runs the testbed. This means that if a new researcher wants to work on the testbed, she would face a steep learning curve. Because I worked on the Strict Priority Queueing detection, I had a need to improve the documentation on existing implementation of SPQ detection. In this paper and through a presentation to fellow students at CSUN, I have conveyed my findings about the details and paraments of testbed. A very beneficial future work for this testbed is cleaning up the testbed and making it easier to run for non-educated users.

In addition, Planet Lab nodes are becoming obsolete. The operating system on these nodes are old and cannot run modern C++ code. To make matters worse, most Planet Lab nodes have become unresponsive. In the future, we need to move away from Planet Lab and start working on newer technology. One alternative to Planet Lab is GENI. The process of updating and documenting the testbed requires a team of students. However, I believe we can achieve this goal in no time.

In addition, I believe we need to update the Strict Priority Queueing detection code. Currently, after the saturation phase is over, we start sending our packets starting with packet ID number 1. The issue here is that the packet numbers are shared between the High and Low packets. This creates some level of confusion while cleaning up the data. The high priority packets should have their own counter. The same is true for low priority ones. My proposed numbering convention can improve feature selection and detection accuracy.

In the future, I would like to consider sending smaller packet trains and extract more features regarding the placement of High and Low priority packets. Based on my findings, I believe there are more patterns that can be discovered in our data that can lead to accurate detections with smaller packets.

Another great future project to consider is to calculate the number of inversions and deletions that converted the original pattern into the received ones. In this experiment, we can look at the original pattern of high and low priority packets as a string. Then, we can create a similar string for each received packet train. An inversion would be the number of switches a packet would have made to move from the original pattern to its new spot on the received pattern. A deletion would be if a packet was lost during transmission and does not exist on the received pattern. Each inversion and deletion would have a penalty value. We can add all the penalty values and compare them between baseline and SPQ packet trains. This comparison could possibly lead to more discoveries about the previously collected data. I did not consider this method because my goal was to divide the packet trains. I wanted to avoid looking at data packets as only one unit.

Additionally, I would like to collect more data and analyze them based on time of the day. I would like to train 24 classifiers for each hour of the day and observe whether the detection accuracy would benefit from this classification. To make this happen, I would need a lot more data. In addition, collecting more data can help us with utilizing other Learning algorithms such as Deep Neural Network.

Ultimately, my goal is to create a light and simple to use browser based detection testbed. The new testbed can be run by any user around the globe. We can do separate analyses for each region. However, this goal can only be achieved if efficiency of our testbed is improved and the network intrusion has been reduced. Everyday users will not have the patience to install a testbed on their own and wait for days before they have collected some relevant data.

Chapter 9: Conclusion

In this chapter, I will summarize my findings regarding the Network Neutrality trainer that I developed over the past academic year. Detecting the violations of net neutrality has been a hot pursuit by the research community [6]. There are many successful methods of detection. Each method has its own advantages and disadvantages. Most detections methods focus on one or two features. My goal was to extract multiple features and look for patterns in data. I have demonstrated that Machine Learning algorithms can generate accurate classifiers.

I believe my results here have indicated that taking multiple features into account can be beneficial for detection mechanisms. As Quality of Service policies evolve, the research community needs to improve their detection classifiers as well. The QoS policies can change and become more elusive to current detection methods. However, if more features are taken into consideration, new detectors can become immune to these changes.

I believe that my work in this paper has indicated that we can send smaller packet trains and reduce the intrusion we have on network. As we increase the number of our features, we could reduce the size of packets that we extract these features from. At the same time, the Machine Learning algorithms can increase the accuracy of our detection with smaller packet sizes. I hope that future researchers in this field would follow our findings and employ Machine Learning algorithms in their detection.

Finally, I would like to point out again that researchers who want to analyze any data should dedicate a majority of their time to understand their data. Data needs to be

sanitized and studied. Multiple efforts are needed to extract features from and find intricacies in data. Even after finding promising results, data should be studied further to extract even more important features.

References

[1] M. B. Tariq, M. Motiwala, N. Feamster and M. Ammar, "Detecting General Network Neutraliry Violations with Causal Inference," *Proceedings of the 5th International Conference,* pp. 289-300, 2009.

[2] Y. Zhang, Z. M. Mao and M. Zhang, "Ascertaining the Reality of Network Neutrality Violation in BAckbone ISPs," in *In Proc. 7th ACM Workshop on Hot Topics in Networka*, 2008.

[3] M. Rahimi and V. Pournaghshband, "An improvement mechanism for low priority traffic TCP performance in Strict Priority Queueing," *2016 International Conference on Computer Communication and Informatics (ICCCI),* pp. 1-5, 2016.

[4] Z. Zhang, O. Mara, Argyraki and Katerina, "Network Neutrality Inference," in *ACM Conference on SIGCOM*, New York, 2014.

[5] M. B. Traiq, M. Motiwala and N. Feaster, "NANO: Network Access Neutrality Observatory," Georgia Institute of Technology, 2008.

[6] P. Kanuparthy and C. Dovrolis, "DiffProbe: Detecting ISP Service Discrimination," in *IEEE INFOCOM*, 2010 Proceeding.

[7] P. Kirth, "Network Neutrality Violation Detection: Methodology, Testbed, Measurement, and Tools," Northridge, 2016.

[8] R. Chang, M. Rahimi and V. Pournaghshband, "Simulating Strict Priority Queueing, Weighted Round Robin, and Weighted Fair Queueing wit NS-3," Advanced Network and Security Research Laboratory Computer Science Department California State University, Northridge, Northridge, 2015.

[9] V. Pournaghshband, "End-to-End Detection of Third-Party Middlebox Interference," 2014.

[10] V. Pournaghsband, A. Afanasyev and P. Reiher, "End-toEnd Detection of Compression Traffic Flows by Intermediaries," *2014 IEEE Network Operations and Management Symposiom (NOMS),* pp. 1-8, May 2014.

[11] D. Li, F. Tian, M. Zhu, L. Wang and L. Sun, "A Novel Framework for Analysis of Global Network Neutrality Based on Packet Loss Rate," *2015 International Conference on Cloud Computing and Big Data (CCBD),* pp. 297-304, 2015.

[12] P. Domingos, "A few Useful Things to Know about Machine Learning," in *Communications of the ACM 55(10),* 2012.

[13] C. Cortes and V. Vapnik, "Support-Vector Network," *Machine Learning,* pp. 20: 273-297, 1995.

[14] D. Frakdin and I. Muchnik, "Support Vector Machines for Classification," *DIMACS Series Discrete Math. Theoretical Comput. Sci., Vol 70,* pp. 13-20, 2006.

[15] "Documentation of scikit-learn 0.18," scikit-learn.org, [Online]. Available: http://scikit-learn.org/stable/documentation.html. [Accessed 10 04 2017].

[16] "SciPy Documentation," SciPy, [Online]. Available: https://www.scipy.org/docs.html.

[17] G. Biau, "Analysis of a Random Forests Model," *Journal of Machine Learning Research,* pp. 13: 1063-1095, 2012.

[18] T. Mitchell, "Decision Tress," in *Machine Learning,* McGraw Hill, 1997, pp. 52-80.

[19] "Quickstart Tutorial," SciPy, [Online]. Available: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html. [Accessed 15 04 2017].

[20] SciKit Learn, "sklearn.model_selection.train_test_split," scikit-learn, [Online]. Available: http://scikit-

learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. [Accessed 25 04 2017].

[21] "sklearn.svm.LinearSVC," scikit-learn, [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html. [Accessed 27 04 2017].

[22] "sklearn.metrics.accuracy_score," scikit-learn, [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html. [Accessed 28 04 2017].

[23] "sklearn.metrics.confusion_matrix," scikit-learn, [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html. [Accessed 28 04 2017].

[24] "sklearn.ensemble.RandomForestClassifier," scikit-learn, [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. [Accessed 29 04 2017].

[25] "1.11. Ensemble methods," SciKit Learn, [Online]. Available: http://scikit-learn.org/stable/modules/ensemble.html.

[26] "Feature importances with forests of trees," scikit-learn, [Online]. Available: http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html. [Accessed 29 04 2017].

[27] P. Kanuparthy and C. Dorvolis, "ShaperProbe: End-to-end Detection of ISP Traffic Shaping using Active Methods," *proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, pp. 473-482.

[28] A. M. Kakhki, A. Razaghpanah, A. Li, H. Koo, R. Golani, D. Choffnes, P. Gill and A. Mislove, "Identifying Traffic Differentiation in Mobile Networks," in *Procedings of the 2015 Internet Measurement Conference*, Tokyo, 2015.